Exercise 1.

Programming a two dimensional cellular automaton - Game of Life. Cellular automata are often used to simulate real-world scenarios. Being rude one could say that weather forecast is just a three dimensional cellular automaton with a bunch of simple rules. The Game of Life is a particular cellular automaton consisting of a two dimensional grid. The grid in turn consists of fields. Each field to which we refer to as a cell can be in one of the two states: the state 1 (alive) or 0 (dead). This automaton is called Game of Life because the state of a cell might change from one generation to another according to some rules. The Game of Life allows arbitrary many generation that are computed iteratively.

The rules for computing the next generation are as follows (each cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent):

1. A dead cell becomes alive if exactly 3 of its adjacent cells are alive.
2. A living cell dies if fewer than 2 or more than 3 of its adjacent cells are alive.
3. In any other case the state of a cell remains the same.

As all cells positioned right at the boarders of the grid do not have 8 adjacency cells, you do not have to consider them for the purpose of simplicity. Consider that when you iterate the cells of the current
generation in order to compute the next generation. Use two nested vectors of boolean variables to store
the state of a grid as shown in the code snippet below.

```cpp
// this generates an 8 x 10 grid
vector<vector<bool>> grid(8, vector<bool>(10, 0));
// printing the grid
for (auto &row : grid) {
    for (auto &cell : row)
        cout << cell << " ";
    cout << endl;
}
// single cells can be accessed by using operator[] two times
cout << "grid at position [1][2] is: " << grid[1][2] << endl;
```

You might like to have a look at [https://en.wikipedia.org/wiki/Conway’s_Game_of_Life](https://en.wikipedia.org/wiki/Conway’s_Game_of_Life) for a
more detailed description.

a) Implement a function `vector<vector<bool>> read_grid(const string &filename);` that reads a a grid
from a text file and parses it into a vector of vector’s of bool variables. (4 P.)

b) Next, implement another function

```cpp
vector<vector<bool>> game_of_life(const vector<vector<bool>>& grid, const size_t N);
```

that returns a grid obtained by ”waiting” (computing) N generations for the input grid. Hint: Use two
temporary ”grid” variables. Compute the cells for the next generation grid by checking the rules
for the current generation and write new states to the next generation grid. When you have com-
pleted the computation for the next generation, use the vector member function `swap()` in order to
swap the contents of the two temporary variables and proceed until you have computed the N-th
generation and return it. (8 P.)

c) You know comes next, implement a function

```cpp
void write_grid(const vector<vector<bool>>& grid, const string &filename);
```

that writes a grid to a
text file. (3 P.)

d) Test all of your functions by reading the grid from `initial_grid.txt`
([https://www.hni.uni-paderborn.de/fileadmin/Fachgruppen/Softwaretechnik/Lehre/CPP_Programming/WS2017_2018/initial_grid.txt](https://www.hni.uni-paderborn.de/fileadmin/Fachgruppen/Softwaretechnik/Lehre/CPP_Programming/WS2017_2018/initial_grid.txt)) that looks a bit like a snowman and compute the grid that is obtained
by waiting (computing) the 10-th generation. Write the result into a file. (1 P.)

**Exercise 2.**

This is an optional exercise that is worth the equivalent of 16 points:

In this exercise you will implement a simple hash table. Do not worry, we will split this task into little
subtasks. A hash table $H$ is a data structure that stores values $v_i$ that are associated with keys $k_i$. We
assume that the keys are unique. A value can be accessed quickly in $H$ by applying a hash function
$h: S \rightarrow \mathbb{N}$ to a key (with $S$ the set of all possible strings which is our key domain for this exercise). $h(k)$
tells us where the value that corresponds to the key $k \in S$ is stored in memory. Since the application of a
hash function on a key is a task that only needs a constant amount of time, $H$ is a data structure allowing
the access of arbitrary values in constant time on average. Thus a hash table is one of the most used data
structures in practice.

a) First you have to provide some code that should make up your hash table $H$. It is probably a good
idea to make $H$ a class, since $H$ is a more sophisticated data type. Do so and create a class `HTable`
(1 P.)
b) In this exercise we want to restrict ourselves to only associate `string` variables to variables of an arbitrary type. For that reason, make `HTable` a class template receiving one template parameter `T`. (1 P.)

c) In order to store the elements in `HTable` in an easily accessible manner provide a data member `vector<pair<string, T>> data` that can store key-value pairs. Additionally provide a data member `vector<bool> used_positions` to keep track of the used positions in your hash table. (1 P.)

d) Provide a constructor `HTable(size_t size);` that initializes the member variable `data` to hold `size` elements and `used_positions` accordingly. (1 P.)

e) Now you need to provide a function \( h \) that receives a key (a `string` variable in our case) and returns a positive integer that shall be the index/position where the key-value pair associated with that key shall be stored. Use the hash function shown in code listing 1 to do the job. The function turns a `string` variable into a natural number \( n \in \{0, \ldots, \text{size} - 1 \} \) that is exactly what we need. (1 P.)

f) Next you can implement a member function `bool insert(const string &key, const T &value);`. `insert()` has to compute the index where to store the key-value pair by using the previously implemented function `hash()`. Insert a key-value pair at the position provided by calling `hash()` on `insert`’s parameter `key` and set the corresponding bit in `used_positions` to mark this position as used. At last let `insert()` return the boolean value `false`. (1 P.)

g) We have made a mistake. What if two keys by accident map to the same index? This is called a hash collision and the probability of such a collision grows with the number of entries in the hash table. One needs a strategy to solve this problem. We resolve this problem by making \( H \) a hash table with so-called "linear probing". That is, we check if the calculated position in `data` is empty. If it is empty insert the key-value pair at this very position. If the computed position is not empty, linearly try if one of the next positions in `data` is empty and insert the data into the next empty position. Let the `insert()` function return `true` when you have to use linear probing when inserting a key-value pair. You are right: As a hash table gets filled up with more and more data you have to do more and more linear probing because of hash collisions. The access behavior of a hash table slowly changes from constant time to linear time in the worst case. If you have reached the end of `data` and still have not found an empty position, continue checking empty positions starting at the beginning. If there is no empty position at all throw a `runtime_error` exception to notify the user of your hash table that the table is full. (3 P.)

h) Now let us provide a function to get data out of our hash table. Implement `T& get(const string &key);` to hash the key and retrieve the value associated with that key at position \( h(key) \). Again, due to possible hash collision (think of how we have inserted data) it might be possible that the calculated entry does not contain the corresponding value. Therefore, you have to compare `key` (formal parameter of `get()`) with the `key` stored at the calculated position. If both keys match (use `==` to check) we have found the right entry and can return the corresponding value. If the keys do not match, use linear probing and check linearly for the next entries and return the value as soon as you find both keys matching. If you reach the end of the underlying `vector` start at the beginning and throw a `runtime_error` exception if the key is not contained in the hash table at all. (4 P.)

i) Overload `friend operator<<(ostream &os, const HTable &h);` to print all key-value pairs. (1 P.)
j) Implement a function `void erase(const string &key);` that deletes the entry that corresponds to the key `key`. You can delegate this erase to `data`'s member function erase. Do not forget to set the bit in `used_positions` to zero to mark the place as free. (1 P.)

k) At last implement a function `void clear();` that deletes all entries in your hash table. The number of elements that can be stored in `data` should remain the same. (1 P.)

l) **This is an optional task:** Implement a function `void resize(size_t size);` that resizes your hash table to new size `size`. (Hint: You will need a temporary variable, because when the size of `data` changes, the position for a key-value pair will probably change too, due to the implementation of `hash()`.) (0 P.)

```
Listing 1: A hash function that turns out to be quite efficient.

size_t hash(const string& key) {
    size_t hash_val = 5381; // have a nice prime number to start with
    for (const char c : key) {
        hash_val = hash_val * 33 + c;
    }
    // since hash_val is a unsigned type we cannot go negative
    // thus we can just ignore overflows in this case (overflow is well-defined for unsigned types)
    // because we need a value between 0 and size-1, we take the remainder of division by table's size
    return hash_val % data.size();
}
```

Now relax, you have done a really good job so far. I wish all of you and your families all the best for Christmas and a happy new year.